A Practical Polynomial Calculus for Arithmetic Circuit Verification

Daniela Ritirc, Armin Biere, Manuel Kauers

Johannes Kepler University, Linz, Austria

Abstract. Generating and automatically checking proofs independently increases confidence in the results of automated reasoning tools. The use of computer algebra is an essential ingredient in recent substantial improvements to scale verification of arithmetic gate-level circuits, such as multipliers, to large bit-widths. There is also a large body of work on theoretical aspects of propositional algebraic proof systems in the proof complexity community starting with the seminal paper introducing the polynomial calculus. We show that the polynomial calculus provides a frame-work to define a practical algebraic calculus (PAC) proof format to capture low-level algebraic proofs needed in scalable gate-level verification of arithmetic circuits. We apply these techniques to generate proofs obtained as by-product of verifying gate-level multipliers using state-of-the-art techniques. Our experiments show that these proofs can be checked efficiently with independent tools.

1 Introduction

Formal verification gives correctness guarantees. However, the process of verification might also not be error-free. A common approach to increase confidence in the results of verification consists of generating machine checkable proofs which are then checked by independent proof checkers. These checkers are less complex than for example theorem provers producing proofs and can also be verified.

For instance many applications of formal verification rely on SAT solvers. Their results can be validated by producing and checking resolution proofs [17,37] or clausal proofs [15,17]. Generating proofs is mandatory in the main track of the SAT Competition since 2016. These approaches have also recently been shown to scale to huge low-level proofs of combinatorial problems such as the Boolean Pythagorean triples problem [18] or Schur Number Five [16].

However, in certain applications, e.g., arithmetic circuit verification, resolution based SAT solving does not work. Especially reasoning about gate-level multipliers is considered to be hard [5]. For arithmetic circuit verification the currently most promising approach uses algebraic reasoning [11,26,30,32].

In this approach each circuit gate is translated into a polynomial to model constraints between its output and inputs, i.e., roots of polynomials are identified as solutions of gate constraints. Additional polynomials ensure that values remain in the Boolean domain. Word-level specifications relating circuit outputs and inputs can also be translated into polynomials. Thus verification boils down to show that the specification polynomial is "implied" by the polynomials induced by the circuit gates (contained in the ideal generated by them).

To validate results of algebraic reasoning the polynomial calculus can be used [12]. It operates on polynomials and allows to check if a polynomial is a logical consequence of a given set of polynomials. The main focus in this area has been on proof complexity to obtain lower-bounds for the degree and size of proofs [20]. For instance [27] introduces a general method to obtain lower bounds and [25] shows that certifying the non-k-colorability of graphs requires proofs of large degree. A more general calculus capable of detecting unsatisfiability of nonlinear equalities as well as inequalities is discussed in [34].

Our paper shows that the polynomial calculus can also be used in practice. In particular we generate low-level algebraic proofs needed to validate the results of ideal membership testing used in arithmetic circuit verification by translating proofs extracted from computer algebra systems to polynomial refutations in the polynomial calculus. After we review preliminaries in Sect. 2, we present a concrete proof format for polynomial calculus proofs, called practical algebraic calculus in Sect. 3. In Sect. 4 we give a comprehensive introduction to arithmetic circuit verification, following [30]. Section 5 introduces the tool flow of verifying and proof checking arithmetic circuits. In our experiments, shown in Sect. 6, our new proof checker PacTrim is used to independently validate the results of multiplier verification [30]. We further apply these techniques to equivalence checking of multipliers [31] and proving certain ring properties, e.g. commutativity of multipliers [3]. In general, we claim that our approach is the first to provide machine checkable proofs for current state-of-the-art techniques in verifying arithmetic circuits [11,26,30,32].

2 Preliminaries

Proof systems are used to validate the results of verification systems. While a verification system only gives a *yes*/*no* answer, a proof system provides additionally a certificate with which the answer can be checked independently. We are concerned here with a proof system for reasoning about polynomial equations. The question is whether the zeroness of a certain set of polynomials implies the zeroness of another polynomial. We consider polynomials $p \in \mathbb{F}[X]$ where $\mathbb{F}$ is a field and $X = \{x_1, \dots, x_n\}$ is a finite set of variables. The function $X \mapsto p(X)$ is called *polynomial function* of p. The *polynomial equation* of p is defined as $p(X) = 0$ and the solutions of this equation are the roots of p. From now on we drop the function argument and write $p = 0$ instead of $p(X) = 0$.

Reasoning with polynomial equations is well-understood both in computer algebra and in computational logic. Already Hilbert and collaborators have studied the theory of polynomial ideals in order to reason about the solution sets of polynomial equations. The application of Gröbner bases [8] by for instance Kapur [21,22,23] has turned the algebraic approach into a valuable computational tool for automated theorem proving with renewed recent interest [1,38].

In order to introduce the notation and terminology needed later, let us give a brief summary of the theory. As far as algebra is concerned, we follow the standard textbooks [4,9,13]. From the logical perspective, we use a variant of the polynomial calculus (PC) as proposed by [12]. It is more flexible than the Nullstellensatz (NS) proof system [2], which is also heavily used in the proof complexity community. The relation between PC and NS in the context of our application is further discussed at the end of this section.

Let $G \subseteq \mathbb{F}[X]$ and $f \in \mathbb{F}[X]$. In logical terms, the question is whether the equation $f = 0$ can be deduced from the equations $g = 0$ with $g \in G$, i.e., every common root of the polynomials $g \in G$ is also a root of f. As we will only consider polynomial equations with right hand side zero, we take the freedom to write f instead of $f = 0$. We write proofs as tuples $P = (p_1, \ldots, p_n)$ of polynomials where each p_i is derived by one of the following rules.

Addition $\dfrac{p_i \qquad p_j}{p_i + p_j}$ p_i, p_j appearing earlier in the proof or are contained in G

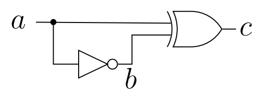
Multiplication $\dfrac{p_i}{qp_i}$ p_i appearing earlier in the proof or is contained in G and $q \in \mathbb{F}[X]$ being arbitrary

If f can be deduced from the polynomials $g \in G$, i.e. $p_n = f$, we write $G \vdash f$. In algebraic terms, $G \vdash f$ means that f belongs to the ideal generated by G. Recall that an *ideal* $I \subseteq \mathbb{F}[X]$ is defined as a set with $0 \in I$ and the closure properties $u, v \in I \Rightarrow u + v \in I$ and $w \in \mathbb{F}[X], u \in I \Rightarrow wu \in I$. If $G = \{g_1, \ldots, g_m\} \subseteq \mathbb{F}[X]$ is a finite set of polynomials, then the ideal generated by G is defined as the set $\{q_1 g_1 + \cdots + q_m g_m : q_1, \ldots, q_m \in \mathbb{F}[X]\}$ and denoted by $\langle G \rangle$. The set G is called a *basis* of the ideal $\langle G \rangle$. It is clear that this is an ideal and that it consists of all the polynomials whose zeroness can be deduced from the zeroness of the polynomials in G. In logical terms we would call G an axiom system and $\langle G \rangle$ the corresponding theory. If we can derive $G \vdash 1$, or in algebraic terms $1 \in \langle G \rangle$, the PC proof is called a *PC refutation*.

Example 1. This example shows that the output c of an XOR gate over an input a and its negation $b = \neg a$ is always true, i.e., $c = 1$ or equivalently $-c + 1\,(= 0)$. We apply the polynomial calculus over the ring $\mathbb{Q}[c, b, a]$. Over $\mathbb{Q}$ a NOT gate $x = \neg y$ is modeled by the polynomial $-x + 1 - y$ and an XOR gate $z = x \oplus y$ is modeled by the polynomial $-z + x + y - 2xy$. Because the variables are of the boolean domain we further need to enforce that every variable can only take the values 0 or 1. Therefore we add for each variable x_i a polynomial of the form $x_i(x_i - 1)$ to the given set of polynomials. The corresponding circuit representation, the given polynomials and a polynomial proof are shown in Fig. 1.

Example 2. Let $G = \{x, x + y\} \subseteq \mathbb{Q}[x, y]$, $f = y$. We have $G \vdash f$. A proof is $P = (-x, y)$. The first entry follows by the multiplication rule from x with $q = -1$, and the second entry follows by the addition rule from the first entry and $x + y$ which is contained in G.

$$\begin{aligned} G = \{ & -b+1-a, \\ & -c+a+b-2ab, \\ & a^2-a,\ b^2-b,\ c^2-c\} \end{aligned}$$



$$\cfrac{\cfrac{\cfrac{-c+a+b-2ab \qquad -b+1-a}{-c+1-2ab} \qquad \cfrac{-b+1-a}{2ab-2a+2a^2}}{-c+1-2a+2a^2} \qquad \cfrac{a^2-a}{-2a^2+2a}}{-c+1}$$

Fig. 1: The circuit, polynomial representation of the gates and proof for Ex. 1.

Thanks to the theory of Gröbner bases [4,8,13], the polynomial calculus is decidable, i.e., there is an algorithm, which for any finite $G \subseteq \mathbb{F}[X]$ and $f \in \mathbb{F}[X]$ can decide whether $G \vdash f$ or not. A basis of an ideal I is called a Gröbner basis if it enjoys certain structural properties whose precise definition is not relevant for our purpose. What matters are the following fundamental facts:

- There is an algorithm (Buchberger's algorithm) which for any given finite set $B \subseteq \mathbb{F}[X]$ computes a Gröbner basis for the ideal $\langle B \rangle$ generated by B.
- Given a Gröbner basis G, there is a computable function $\mathrm{red}_G \colon \mathbb{F}[X] \to \mathbb{F}[X]$ such that $\forall\, p \in \mathbb{F}[X] : \mathrm{red}_G(p) = 0 \iff p \in \langle G \rangle$.
- Moreover, if $G = \{g_1, \ldots, g_m\}$ is a Gröbner basis of an ideal I and $p, r \in \mathbb{F}[X]$ are such that $\mathrm{red}_G(p) = r$, then there exist $h_1, \ldots, h_m \in \mathbb{F}[X]$ such that $p - r = h_1 g_1 + \cdots + h_m g_m$, and such polynomials h_i can be computed.

Consider the extended calculus with the additional rule

$$\text{Radical} \quad \frac{p_i^m}{p_i} \quad \begin{array}{l} m \in \mathbb{N} \setminus \{0\} \text{ and} \\ p_i^m \text{ appearing earlier in the proof or is contained in } G. \end{array}$$

If the polynomial f can be deduced from the polynomials g, where $g \in G$, with the rules of PC and this additional radical rule, we write $G \vdash^+ f$ and call this proof *radical proof* $(\vdash^+)$. In algebra, the set $\{\, f \in \mathbb{F}[X] : G \vdash^+ f \,\}$ is called the *radical ideal* of G and is typically denoted by $\sqrt{\langle G \rangle}$.

Also the extended calculus $\vdash^+$ is decidable. It can be reduced to $\vdash$ using the so-called Rabinowitsch trick [13, 4§2 Prop. 8], which says

$$f \in \sqrt{\langle G \rangle} \iff 1 \in \langle G \cup \{yf - 1\} \rangle \quad \text{or} \quad G \vdash^+ f \iff G \cup \{yf - 1\} \vdash 1,$$

depending whether you prefer algebraic or logic notation. In both cases, y is a new variable and the ideal/theory on the right hand sides is understood as an ideal/theory of the extended ring $\mathbb{F}[X, y]$. The Rabinowitsch trick is therefore used to replace a radical proof $(\vdash^+)$ by a PC refutation.

For a given set $G \subseteq \mathbb{F}[X]$, a *model* is a point $u = (u_1, \ldots, u_n) \in \mathbb{F}^n$ such that for all $g \in G$ we conclude that $g(u_1, \ldots, u_n) = 0$. Here, by $g(u_1, \ldots, u_n)$ we mean the element of $\mathbb{F}$ obtained by evaluating the polynomial g for $x_1 = u_1, \ldots, x_n = u_n$. For a set $G \subseteq \mathbb{F}[X]$ and a polynomial $f \in \mathbb{F}[X]$, we write

$G \models f$ if every model for G is also a model for $\{f\}$. Given $G \subseteq \mathbb{F}[X]$, define $V(G)$ as the set of all models of G. For an algebraically closed field $\mathbb{F}$, Hilbert's Nullstellensatz [13, 4§1 Thms. 1 and 2] asserts that $V(G)$ is nonempty if and only if $1 \notin \langle G \rangle$, and furthermore, $f \in \sqrt{\langle G \rangle} \iff V(G) \subseteq V(\{f\})$. In other words, $G \models f \iff G \vdash^{+} f$. Particularly, the PC including the radical rule is correct ("$\Leftarrow$") and complete ("$\Rightarrow$"). In combination with Rabinowitsch's trick, we can therefore decide the existence of models and furthermore produce certificates for the non-existence of models.

For our applications, only models $u \in \{0,1\}^n \subseteq \mathbb{F}^n$ matter. Let us write $G \models_{\text{bool}} f$ if every model $u \in \{0,1\}^n$ of G is also a model of $\{f\}$. Using basic properties of ideals as described in [13, 4§3 Thm. 4], it is easy to show that $G \models_{\text{bool}} f \iff G \cup B \models f$, where $B = \{x_i(x_i - 1) : i = 1, \ldots, n\}$. Furthermore, the equivalence $G \cup B \models f \iff G \cup B \vdash^{+} f$ holds also when $\mathbb{F}$ is not algebraically closed, because changing from $\mathbb{F}$ to its algebraic closure $\bar{\mathbb{F}}$ will not have any effect on the models in $\{0,1\}^n$. Finally, let us remark that the finiteness of $\{0,1\}^n$ also implies that $G \cup B \vdash^{+} f \iff G \cup B \vdash f$. This follows from Seidenberg's lemma [4, Lemma 8.13] and generalizes Theorem 1 of [12].

In contrast to a PC refutation $G \cup \{1 - yf\} \cup B \vdash 1$, where each polynomial in the proof is generated using the rules of PC, a refutation in the NS proof system is a set of polynomials $Q = \{q_1, \ldots, q_m\} \subseteq \mathbb{F}[X]$ such that

$$\sum_{i=0}^{m} q_i p_i = 1 \quad \text{for} \quad p_i \in G \cup \{1 - yf\} \cup B.$$

Although both systems are able to verify correctness of a refutation, we will use PC and not the NS proof system, because for arithmetic circuit verification we will rewrite some polynomials of $G \cup \{1 - yf\} \cup B$, and thus gain an optimized algebraic representation of the circuit, cf. Sect. 4. In a correct NS refutation we would also need to express these rewritten polynomials as a linear combination of elements of $G \cup \{1 - yf\} \cup B$ and thus lose the optimized representation, which will most likely lead to an exponential blow-up of monomials in the NS proof [10]. In PC we can generate these optimized polynomials on-the-fly and then use these polynomials to show the correctness of the refutation.

3 Practical Algebraic Calculus

For practical proof checking we translate the abstract polynomial calculus (PC) into a concrete proof format, i.e., we only define a format based on PC, which is logically equivalent but more precise. In principle a proof in PC can be seen as a finite sequence of polynomials derived from given polynomials and previously inferred polynomials by applying either an addition or multiplication rule.

To ensure correctness of each rule it is of course necessary to know which rule was used, to check that it was applied correctly, and in particular which given or previously derived polynomials are involved. During proof generation these polynomials are usually known and thus we require that all of this information

```
     letter  ::=  ‘a’ | ‘b’ | ... | ‘z’ | ‘A’ | ‘B’ | ... | ‘Z’
     number  ::=  ‘0’ | ‘1’ | ... | ‘9’
   constant  ::=  (number)+
   variable  ::=  letter (letter | number)*
      power  ::=  variable [‘^’ constant]
       term  ::=  power (‘*’ power)*
   monomial  ::=  constant | [constant ‘*’] term
   operator  ::=  ‘+’ | ‘-’
 polynomial  ::=  [‘-’] monomial (operator monomial)*
      given  ::=  (polynomial ‘;’)*
       rule  ::=  (‘+’ | ‘*’) ‘:’ polynomial ‘,’ polynomial ‘,’ polynomial ‘;’
      proof  ::=  (rule ‘;’)*
```

Fig. 2: Syntax of given polynomials and proofs in PAC-format

is part of a rule in our concrete *practical algebraic calculus* (PAC) proof format to simplify proof checking. The syntax of PAC is shown in Fig. 2. White space is allowed everywhere except between letters and digits in a constant or a variable. A proof rule contains four components

$$o : v, w,\ p;$$

The first component o denotes the operator which is either ‘+’ for addition or ‘*’ for multiplication. The next two components v, w specify the two (antecedent) polynomials used to derive p (conclusion). In the multiplication rule w plays the role of the polynomial q of the multiplication rule of PC, cf. Sect. 2. A refutation in PAC is a proof, which contains a non-zero constant polynomial (typically just the constant “1”) as conclusion p of a rule.

As discussed above we do not need the radical rule for our purpose, even though it could be easily added. Further note that the format is independent of the domain of the models u, e.g., $u \in \{0,1\}^n$ for gate-level circuit verification, to which the values of variables are restricted. If such a restriction is necessary, all elements of the corresponding set B (often also called field polynomials) have to be added to the given set of polynomials.

Although the definition of number together with the definition of polynomial only allows integer coefficients this is not a severe restriction. Rational number coefficients can be simulated by multiplying involved polynomials with appropriate non-zero constants to eliminate denominators.

Example 3. Consider again Ex. 1. To test membership of $1 - c \in \sqrt{\langle G \rangle}$ we add $1 + y(c-1)$ to the set of given polynomials G in order to apply Rabinowitsch's trick and obtain a PAC refutation:

```
+ : -c+a+b-2a*b,  -b+1-a,       -c+1-2a*b;
* : -b+1-a,       -2a,          2a*b-2a+2a^2;
+ : -c+1-2a*b,    2a*b-2a+2a^2, -c+1-2a+2a^2;
* : a^2-a,        -2,           -2a^2+2a;
+ : -c+1-2a+2a^2, -2a^2+2a,     -c+1;
* : -c+1,         y,            -c*y+y;
+ : -c*y+y,       1+c*y-y,      1;
```

```
input   $G$              sequence of given polynomials
        $r_1 \cdots r_k$   sequence of PAC proof rules

output  "incorrect", "correct-proof", or "correct-refutation"

$P_0 \leftarrow G$
for $i \leftarrow 1 \ldots k$
    let $r_i = (o_i, v_i, w_i, p_i)$
    case $o_i = +$
        if $v_i \in P_{i-1} \wedge w_i \in P_{i-1} \wedge p_i = v_i + w_i$ then $P_i \leftarrow \text{append}(P_{i-1}, p_i)$
        else return "incorrect"
    case $o_i = *$
        if $v_i \in P_{i-1} \wedge p_i = v_i * w_i$ then $P_i \leftarrow \text{append}(P_{i-1}, p_i)$
        else return "incorrect"
for $i \leftarrow 1 \ldots k$
    if $p_i$ is a non zero constant polynomial then return "correct-refutation"
return "correct-proof"
```

Fig. 3: Proof Checking Algorithm

For proof validation we need to make sure that two properties hold. The *connection property* states that the components v, w are either given polynomials or conclusions of previously applied proof rules. For multiplication we only have to check this property for v, because w is an arbitrary polynomial. By the second property, called *inference property*, we verify the correctness of each rule, namely we simply calculate $v+w$ resp. $v*w$ and check that the obtained result matches p. In a correct PAC refutation we further need to verify that at least one p_i is a non-zero constant. The complete checking algorithm is shown in Fig. 3.

4 Circuit verification using Computer Algebra

Following [11,30,31,32,33,36] we consider gate-level (integer) multipliers with $2n$ input bits $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1} \in \{0,1\}$ and $2n$ output bits $s_0, \ldots, s_{2n-1} \in \{0,1\}$. Each internal gate (output) is represented by a further variable $l_1, \ldots, l_m$. In this setting let $X = a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, l_1, \ldots, l_m, s_0, \ldots, s_{2n-1}$. Then a multiplier is correct iff for all possible inputs the following specification holds:

$$\sum_{i=0}^{2n-1} 2^i s_i = \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \tag{1}$$

Using algebraic reasoning this can be verified by showing that the specification is contained in the ideal generated by the gate constraints. For each logical gate in the circuit a so-called *gate polynomial* $g \in \mathbb{Q}[X]$ representing the relation between the gate inputs and output is defined. Example 1 defines these polynomials for a NOT and an XOR gate. Indicating that the circuit operates over Boolean variables we add for each variable $x_i \in X$ the relation $x_i(x_i - 1)$ matching the definition of B in the last paragraph of Sect. 2 to the gate polynomials G.

Although all variables are restricted to boolean values we use $\mathbb{Q}$ as the base field. Using $\mathbb{Q}$ connects the circuit specification (Eqn. (1)) to multiplication in $\mathbb{Q}$. The specification would be the same over $\mathbb{Z}$, but $\mathbb{Z}$ is not a field, hence the underlying Gröbner basis theory would be more complex. Theoretically reasoning in the field $\mathbb{Z}_2$ is possible, but probably would be much more involved. A more precise comparison will be done in the future.

A term order is a *lexicographic term order* if for all terms $\sigma_1 = x_1^{u_1} \cdots x_n^{u_n}$, $\sigma_2 = x_1^{v_1} \cdots x_n^{v_n}$ we have $\sigma_1 < \sigma_2$ iff there exists i with $u_j = v_j$ for all $j < i$, and $u_i < v_i$. If the terms in the gate polynomials are ordered according to such a lexicographic variable ordering where the variable corresponding to the output of a gate is always bigger than the variables corresponding to inputs of the gate, then by Buchberger's product criterion [13] the gate polynomials define a Gröbner basis for the ideal generated by the gate polynomials. Thus the correctness of the circuit can be shown by reducing the specification by the gate polynomials using polynomial reduction (red_G) and checking if the result is zero. We generate and check proofs for this reduction, cf. Sect. 5.

Directly reducing the specification without rewriting the Gröbner basis leads to an explosion of intermediate results [30]. In practice it is necessary to use rewriting techniques to simplify the Gröbner basis. In recent work [32] a reduction scheme was proposed which effectively (partially) reduces the Gröbner basis. These preprocessing steps [32] are also applied in [30], where we introduced a column-wise checking algorithm which cuts the circuit into $2n$ slices S_i with $0 \leq i < 2n$ such that each slice contains exactly one output bit s_i. In each slice the relation that the sum of the outgoing carries C_{i+1} and the output-bit s_i is equal to the sum of the partial products $P_i = \sum_{k+l=i} a_k b_l$ and the incoming carries of the slice C_i has to hold. Thus we define for each slice S_i a corresponding specification $C_i = 2C_{i+1} + s_i - P_i$. Initially we set $C_{2n} = 0$ and recursively calculate C_i as the remainder of reducing $2C_{i+1} + s_i - P_i$ by the gate polynomials of the corresponding slice. In a correct multiplier $C_0 = 0$ has to hold. Hence each slice is verified recursively, thus the problem of circuit verification is divided into smaller more manageable sub-problems.

In [31] we further improved incremental checking by eliminating variables [7], local to full- and half-adders. Since these preprocessing and incremental algorithms are complex and error prone to implement but essential to achieve scalable verification we also generate and check proofs for them.

5 Engineering

We take as input circuit an And-Inverter Graph (AIG) [24] in the common AIGER format [6]. The AIG is then verified using the computer algebra system Mathematica [35]. We also generate proofs in our PAC-format (c.f. Sect. 3) which then are either passed on to the computer algebra system Singular [14] or to our own algebraic proof checker PacTrim. The complete verification flow is depicted in Fig. 4. Boxes with ".⟨suffix⟩" refer to the input AIG or generated files. The variable n defines the length of the two input bitvectors of the multiplier.

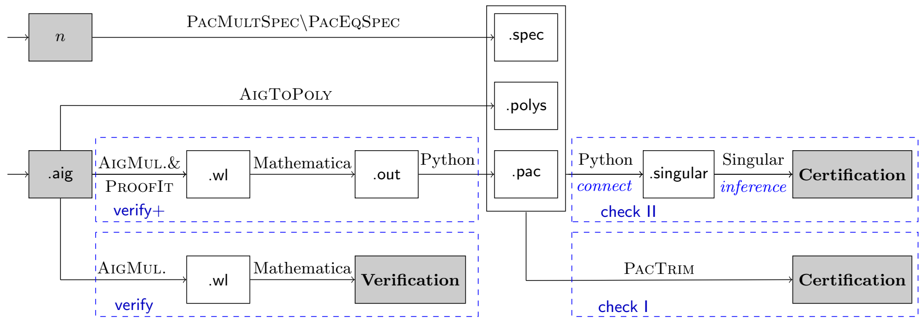


Fig. 4: Toolflow of verifying and proof checking circuits

The tool AigMulToPoly [30,31] is used for verification without generating proofs (verify). It takes an AIG as input and produces a file which can be passed on to either Mathematica or Singular, which then performs the actual ideal membership test. Different option settings can be selected to enable or disable the preprocessing and rewriting techniques discussed in Sect. 4.

For proof generation (verify+) we use a second tool ProofIt which takes the output file from AigMulToPoly as well as the original AIG and returns a file which can be passed on to Mathematica. In Mathematica the proof (.pac) is calculated. In the tool AigToPoly the original AIG is translated into a set of polynomials G without applying any preprocessing. Together with the set $B = \{x_i(x_i - 1) \mid x_i \in X\}$ these polynomials define the given set of polynomials $G \cup B$ of the PAC proof (.polys). This is a rather trivial task implemented in less than 130 lines of C code (half of them are just about command line option handling) using the AIGER [6] library for parsing.

In the same spirit PacMultSpec and PacEqSpec have been implemented to produce the specifications we want to verify (.spec). In PacMultSpec we simply generate the multiplier specification as given in Sect. 4, i.e. Eqn. (1) flattened. In PacEqSpec we generate a similar specification for equivalence checking of two multipliers [31]. To gain a PAC refutation both types of specifications are produced in negated form using the Rabinowitsch trick and hence become part of the given set of polynomials.

Each polynomial of AigMulToPoly which is derived during preprocessing needs to be checked if it is a logical consequence of the given set of polynomials. Hence for each preprocessed polynomial f the representation modulo the given set of polynomials $G \cup B = \{g_1, \ldots, g_k\}$ is calculated in Mathematica using the built-in function "PolynomialReduce". This command does not only allow to compute the reduction $\text{red}_{G\cup B}(f) = r$, but it also returns cofactors $h_1, \ldots, h_k$ such that $f = h_1 g_1 + \ldots + h_k g_k + r$. If the preprocessing is done correctly the derived polynomials f are contained in the ideal $\langle G \cup B \rangle$, thus $\text{red}_{G\cup B}(f) = 0$ and the above representation simplifies to $f = h_1 g_1 + \ldots + h_k g_k$. Knowing the cofactors h_i and the corresponding elements of $G \cup B$ we generate proof rules in PAC in the following way. First we generate a multiplication proof rule for each product $h_i g_i$.

$$* : g_1, h_1, h_1 g_1; \qquad \cdots \qquad * : g_k, h_k, h_k g_k;$$

In the listed rules the result p is always depicted simply as the product $h_i g_i$, but in the actual PAC proof p is written in expanded (flattened) form. These products are now simply added together as follows:

$$\begin{array}{lll}
+ : h_1 g_1, & h_2 g_2, & h_1 g_1 + h_2 g_2; \\
+ : h_1 g_1 + h_2 g_2, & h_3 g_3, & h_1 g_1 + h_2 g_2 + h_3 g_3; \\
 & \vdots & \\
+ : h_1 g_1 + \ldots + h_{k-1} g_{k-1}, & h_k g_k, & f;
\end{array}$$

In the experiments we also use a non-incremental verification approach where we do not use the incremental optimizations presented in Sect. 4, hence we have to reduce the complete word-level specification of a multiplier by the (preprocessed) gate and field polynomials. Extracting a proof works in the same way as just described for the preprocessed polynomials.

Generating proofs for incremental verification is also similar, but instead of the word-level specification of the multiplier we have to use the *incremental specifications* $C_i = 2C_{i+1} + s_i - P_i$ of each slice, cf. Sect. 4. The polynomials C_i describing the incoming carries of a slice can be derived by calculating $\text{red}_{G \cup B}(2C_{i+1} + s_i - P_i) = C_i$. Since verification can be assumed to succeed we have $C_{2n} = 0$ and $C_0 = 0$. As described in the last bullet on fundamental facts in Sect. 2 we are able to obtain cofactors $h_1, \ldots, h_k$ such that $2C_{i+1} + s_i - P_i - C_i = h_1 g_1 + \ldots + h_k g_k$ and consequently a translation into the PAC-format to derive the left-hand side of the equation.

To derive the word-level specification of a multiplier from the incremental specifications we first multiply for each slice S_i its incremental specification $C_i = 2C_{i+1} + s_i - P_i$ by the constant 2^i.

$$\begin{array}{lll}
* : 2C_1 + s_0 - P_0, & 1, & 2C_1 + s_0 - P_0; \\
* : 2C_2 + s_1 - P_1 - C_1, & 2, & 4C_2 + 2s_1 - 2P_1 - 2C_1; \\
 & \vdots & \\
* : s_{2n-1} - P_{2n-1} - C_{2n-1}, & 2^{2n-1}, & 2^{2n-1} s_{2n-1} - 2^{2n-1} P_{2n-1} - 2^{2n-1} C_{2n-1};
\end{array}$$

Subsequent accumulation of the polynomials above using PAC addition rules cancels the terms C_i and $\sum_{i=0}^{2n-1} 2^i s_i - \sum_{i=0}^{2n-1} 2^i P_i$ remains. It holds that the sum of partial products can be reordered to $\sum_{i=0}^{2n-1} 2^i P_i = (\sum_{i=0}^{n-1} 2^i a_i)(\sum_{i=0}^{n-1} 2^i b_i)$ [30] and thus we are able to deduce the word-level specification of multipliers.

In both approaches the incremental as well as the non-incremental one we multiply the word-level specification of the multiplier by the additional variable y and add it to the given polynomial $1 - y * spec \in G \cup B$ to derive 1 and thus obtain a correct PAC refutation.

As Fig. 4 shows we have two different flows for checking PAC proofs independently from Mathematica, which was used for verification. The first one uses Python scripts to validate the *connection* property of each rule and whether the proof actually defines a refutation. With Singular we check the *inference* property of each proof line, which in essence uses Singular as a calculator for adding and multiplying polynomials.

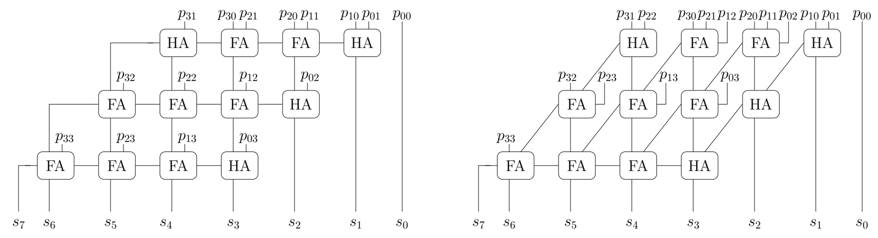

p_{31} p_{30} p_{21} p_{20} p_{11} p_{10} p_{01} p_{00}
HA FA FA HA
p_{32} p_{22} p_{12} p_{02}
FA FA FA HA
p_{33} p_{23} p_{13} p_{03}
FA FA FA HA
s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
p_{31} p_{22} p_{30} p_{21} p_{12} p_{20} p_{11} p_{02} p_{10} p_{01} p_{00}
HA FA FA HA
p_{32} p_{23} p_{13} p_{03}
FA FA FA HA
p_{33}
FA FA FA HA
s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Fig. 5: Architecture of "btor" (left) and "sparrc" (right), where $p_{ij} = a_i b_j$ [31]

We also provide a new dedicated proof checker called PacTrim implemented from scratch in C. It has similar features as DRAT-trim, which is the standard proof checker in the SAT community for clausal proofs (and is used in the SAT Competition – see also [16,18]). Our new PacTrim checker contains a parser for PAC proofs and checks the *connection* property using hash tables and the *inference* property using a dedicated stand-alone implementation of polynomial arithmetic over arbitrary precision integers represented as strings.

While the first approach is rather general and easy to adapt it is, as the experiments confirm, less robust (due to for instance the limit on variables in Singular) and more importantly far less efficient than our dedicated checker. The latter also allows to produce proof cores (of both original polynomials and proof lines), and is also much closer to being certifiable.

6 Experiments

In our experiments we generate and validate PAC proofs for the (integer) multiplier benchmarks used in [30,31]. The "btor"-benchmarks are generated by Boolector [28] and the "sparrc"-multipliers are part of the bigger AOKI benchmark set [19], containing several multiplier architectures. In both multiplier architectures the partial products are generated as products of two input bits which are then accumulated by full- and half-adders, as shown in Fig. 5 for input size $n = 4$. In "btor"-multipliers the full- and half-adders are accumulated in a grid-like structure, thus they are considered as array multipliers, whereas in "sparrc"-multipliers full- and half-adders are accumulated diagonally.

In all our experiments we use a standard Ubuntu 16.04 Desktop machine with Intel i7-2600 3.40GHz CPU and 16 GB of main memory. The (wall-clock) time limit is 90 000 seconds and the main memory usage is limited to 7GB. The time in our experiments is measured in seconds (wall-clock time). We mark unfinished experiments by TO (reached time limit), MO (reached memory limit) or by EE, when an error state is reached. An error state is reached by Singular, because it has a limit of 32767 on the number of ring variables. All experimental data, benchmarks and source code is available at `http://fmv.jku.at/pac`.

In Table 1 we separately list the time taken for verification, the generation as well as checking of PAC-proofs for "btor"and "sparrc" multipliers of different input bitwidth n. The third column lists configurations of AigMulToPoly. The default configuration uses incremental column-wise slicing of [30], c.f. Sect. 4,

n	mult	option	verify	verify+	chk I	con	inf	chk II	length	core	size	core	deg
4	btor	inc	0	1	0	0	0	0	646	68%	3551	72%	6
4	btor	inc-add	0	1	0	0	0	0	594	62%	4001	63%	5
4	btor	noninc	0	1	0	0	0	0	638	68%	3862	74%	6
8	btor	inc	1	4	0	0	0	0	3350	65%	21169	70%	6
8	btor	inc-add	0	3	0	0	0	0	2914	62%	21915	64%	5
8	btor	noninc	1	5	0	0	0	0	3334	65%	28227	78%	6
16	btor	inc	4	70	0	1	3	4	14998	64%	106853	72%	6
16	btor	inc-add	1	37	0	1	3	4	12738	61%	104351	66%	5
16	btor	noninc	4	78	0	1	9	10	14966	64%	231643	87%	6
32	btor	inc	44	1631	1	26	57	83	63254	64%	533773	76%	6
32	btor	inc-add	7	801	1	18	49	67	53122	61%	487911	69%	5
32	btor	noninc	65	1811	5	29	522	551	63190	64%	2594059	95%	6
64	btor	inc	622	49638	4	586	4539	5125	259606	63%	2839901	81%	6
64	btor	inc-add	121	22378	4	414	4236	4650	216834	61%	2387831	74%	5
64	btor	noninc	MO	MO	-	-	-	-	-	-	-	-	-
4	sparrc	inc	0	1	0	0	0	0	753	64%	4943	68%	6
4	sparrc	inc-add	0	1	0	0	0	0	764	65%	8156	66%	8
4	sparrc	noninc	0	1	0	0	0	0	745	65%	5252	71%	6
8	sparrc	inc	1	8	0	0	0	0	3917	62%	30494	69%	6
8	sparrc	inc-add	0	7	0	0	1	1	3964	63%	59330	63%	8
8	sparrc	noninc	1	33	0	0	0	1	3901	63%	37477	75%	6
16	sparrc	inc	8	134	0	2	6	7	17445	62%	152698	71%	6
16	sparrc	inc-add	1	112	0	2	18	20	17804	63%	317874	62%	8
16	sparrc	noninc	11	2696	0	2	15	17	17413	62%	276885	84%	6
32	sparrc	inc	104	3582	1	43	132	175	73301	62%	735218	74%	6
32	sparrc	inc-add	8	2611	2	55	402	457	75244	63%	1492082	63%	8
32	sparrc	noninc	351	TO	-	-	-	-	-	-	-	-	-
64	sparrc	inc	1575	TO	-	-	-	-	-	-	-	-	-
64	sparrc	inc-add	133	80906	12	1307	EE	EE	309164	62%	6727026	65%	8
64	sparrc	noninc	MO	-	-	-	-	-	-	-	-	-	-

Table 1: Wordlevel proof checking

both with (inc-add) and without (inc) our new optimization of eliminating local variables in full- and half-adders [31]. In the third configuration (noninc) the whole word-level specification is reduced without any slicing of the multiplier.

The time needed for verification, proof generation and proof checking is listed in the following columns. The corresponding execution paths are marked in Fig. 4 by dashed rectangles. The column verify shows the time Mathematica needs to verify the multiplier, column verify+ shows the time needed to generate the proof including the time of verify and in column chk I we measure the time our own proof checker PacTrim needs to validate the proof. The time Python needs to verify the *connection* property is listed in column con and the time Singular needs to verify the *inference* property is listed in column inf. The column chk II is the total time needed to verify the proof with Python and Singular. We did not include the time the tools AigToPoly, PacMultSpec and PacEqSpec need, because in the worst-case it only takes a second for 64-bit multipliers.

Inspired by [27] we also compute and include the number of polynomials in a proof (length), the total number of monomials of the derived polynomials (size), counted with repetition, and the maximum total degree of any monomial (deg).

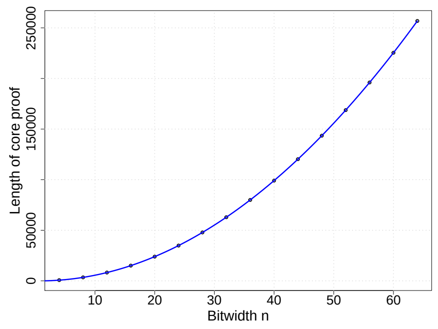
n	mult	verify	verify+	chk I	con	inf	chk II	length	core	size	core	deg
4	btor-btor	1	1	0	0	1	1	1170	59%	7952	61%	5
8	btor-btor	1	6	0	0	1	1	5794	59%	43902	63%	5
16	btor-btor	2	75	1	5	10	14	25410	59%	210666	65%	5
32	btor-btor	27	1632	3	87	189	277	106114	59%	995330	69%	5
64	btor-btor	502	45155	15	1625	EE	EE	433410	59%	4942642	74%	5
4	btor-sparrc	1	2	0	1	1	2	1340	61%	12107	64%	8
8	btor-sparrc	1	9	1	1	2	3	6844	61%	81317	63%	8
16	btor-sparrc	3	148	1	7	42	48	30476	61%	424189	63%	8
32	btor-sparrc	28	3456	7	163	848	1011	128236	60%	1999501	64%	8
4	sparrc-sparrc	1	2	0	0	0	1	1510	62%	16270	65%	8
8	sparrc-sparrc	1	12	1	1	5	6	7894	62%	118820	63%	8
16	sparrc-sparrc	2	223	2	9	73	82	35542	61%	638248	62%	8
32	sparrc-sparrc	29	5363	11	308	1591	1899	150358	61%	3006256	63%	8

Table 2: Equivalence proof checking

Usually not all given polynomials in the data set $G \cup \{1 - yf\} \cup B$ are needed to derive a correct refutation, especially only a small subset of B is used. Thus next to the length and size columns we list the percentage of polynomials and monomials which are actually necessary to derive a PAC refutation (core) w.r.t. the number of original and derived polynomials.

In general it can be seen that "sparrc"-multipliers need more time and space for verification, certification and proof checking than "btor"-multipliers. By far most of the time is needed for generating the proofs. For more scalable proof generation it is clear that computer algebra systems would need to be adapted to support proof generation on-the-fly or even application specific algebraic reasoning engines have to be implemented. Checking the proof with PacTrim takes a fraction of the time needed for verification, at most 12 seconds, even for 64 bit multipliers. Proof checking using an independent computer algebra system takes much longer – for 64 bit multipliers more than 4000 seconds.

In further experiments shown in Table 2 we construct proofs for the commutativity property of multipliers, i.e., we want to prove for a certain multiplier architecture that $A * B = B * A$ holds. Among other things it was shown in the work of [3] that polynomial sized resolution proofs for the commutativity property of array and diagonal multipliers exist. Motivated by this result we generate proofs for these two multiplier architectures, where "btor"-multipliers play the role of array multipliers and "sparrc"-multipliers are considered as diagonal multipliers. We generate the commutativity miters by checking the equivalence of a multiplier and the same multiplier with input bit-vectors swapped (btor-btor, sparrc-sparrc). Furthermore we derive proofs for checking the equivalence of the two architectures "btor" vs. "sparrc" (btor-sparrc). The columns in Table 2 follow the same structure as in Table 1. In all commutativity and equivalence checking experiments we used the configuration "inc-add", which uses our incremental column-wise slicing of [30] with the optimization of eliminating local variables in full- and half-adders. We did not include commutativity or equivalence checking experiments containing "sparrc" multipliers with bit-width $n = 64$, because we reached an error state (EE) in the experiments of Table 1.



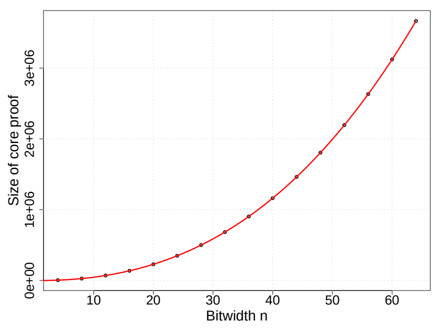


Fig. 6: Length and size of btor-btor commutativity check

In Fig. 6 data points depicting core size (left plot) and core length (right plot) of the "btor-btor"-commutativity proofs are shown for various input bitwidths n. The additional polynomial curves are fitted to the data points (using linear regression with R). For the proof length we used a parameterized model of a quadratic polynomial. The proof size required a cubic polynomial. In both cases the match is perfect, with absolute values of residuals less than $9*10^{-10}$. This empirically suggested quadratic complexity of algebraic proofs compares favourably to the $\mathcal{O}(n^7 \log n)$ upper bound for resolution proofs given in Thm. 2 of [3].

Comparing the meta data of the "btor-btor" and "sparrc-sparrc"-benchmarks the proof lengths of "sparrc-sparrc"-benchmarks are of the same magnitude as the proof lengths of "btor-btor"-benchmarks. The proof sizes of "sparrc-sparrc" are around three times as big as the proof sizes of "btor-btor" with nearly same percentages for the cores. Hence both measurements of "sparrc-sparrc"-benchmarks can also be depicted by quadratic and cubic curves.

7 Conclusion

This paper applies proof checking to algebraic reasoning, not only in theory, but also in practice, in order to validate verification techniques based on computer algebra. We show how the abstract polynomial calculus [12] can be instantiated to yield a practical proof format (PAC). Proofs in this format can be obtained as by-product of verifying multiplier circuits using state-of-the-art techniques and can be checked with our new proof checker tool PacTrim in a fraction of the time needed for verification. Our experiments produce small polynomial proofs which certify the correctness of certain multipliers. The theoretical analysis in [3] gives much larger polynomial upper bounds (for clausal resolution proofs).

To explore the connection between PAC and clausal proof systems, such as RUP and DRAT [17], is an interesting subject for future work, as well as embedding PAC into more general systems, such as Isabelle [29].

We want to thank Thomas Sturm for pointing out the Rabinowitsch trick to the second author and Jakob Nordström for discussions on the polynomial calculus and Nullstellensatz proof systems. This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), Y464-N18, SFB F5004.

References

1. E. Ábrahám, J. Abbott, B. Becker, A. M. Bigatti, M. Brain, B. Buchberger, A. Cimatti, J. H. Davenport, M. England, P. Fontaine, S. Forrest, A. Griggio, D. Kroening, W. M. Seiler, and T. Sturm. Satisfiability checking and symbolic computation. *ACM Comm. Computer Algebra*, 50(4):145–147, 2016.
2. P. Beame, R. Impagliazzo, J. Krajícek, T. Pitassi, and P. Pudlák. Lower bounds on hilbert's nullstellensatz and propositional proofs. In *PROCEEDINGS OF THE LONDON MATHEMATICAL SOCIETY*, pages 1–26, 1996.
3. P. Beame and V. Liew. Towards verifying nonlinear integer arithmetic. In *CAV*, volume 10427 of *LNCS*, pages 238–258. Springer, 2017.
4. T. Becker, V. Weispfenning, and H. Kredel. *Gröbner Bases*. Springer, 1993.
5. A. Biere. Collection of combinational arithmetic miters submitted to the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. Univ. Helsinki, 2016.
6. A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and beyond. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.
7. A. Biere, M. Kauers, and D. Ritirc. Challenges in verifying arithmetic circuits using computer algebra. In *SYNASC*, 2017, in press.
8. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, 1965.
9. B. Buchberger and M. Kauers. Gröbner basis. *Scholarpedia*, 5(10):7763, 2010. `http://www.scholarpedia.org/article/Groebner_basis`.
10. J. Buresh-Oppenheim, M. Clegg, R. Impagliazzo, and T. Pitassi. Homogenization and the polynomial calculus. *Computational Complexity*, 11(3-4):91–108, 2002.
11. M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi. Verification of gate-level arithmetic circuits by function extraction. In *DAC*, pages 52:1–52:6. ACM, 2015.
12. M. Clegg, J. Edmonds, and R. Impagliazzo. Using the groebner basis algorithm to find proofs of unsatisfiability. In *STOC*, pages 174–183. ACM, 1996.
13. D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms.* Springer, 1997.
14. W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 4-1-0. `http://www.singular.uni-kl.de`, 2016.
15. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 10886–10891. IEEE Computer Society, 2003.
16. M. J. H. Heule. Schur number five. *CoRR*, abs/1711.08076, 2017.
17. M. J. H. Heule and A. Biere. Proofs for satisfiability problems. In *All about Proofs, Proofs for All*, volume 55, pages 1–22, 2015.
18. M. J. H. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.
19. N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal design of arithmetic circuits based on arithmetic description language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.
20. R. Impagliazzo, P. Pudlák, and J. Sgall. Lower bounds for the polynomial calculus and the gröbner basis algorithm. *Computational Complexity*, 8(2):127–144, 1999.
21. D. Kapur. Geometry theorem proving using hilbert's nullstellensatz. In *SYMSAC*, pages 202–208. ACM, 1986.
22. D. Kapur. Using gröbner bases to reason about geometry problems. *J. Symb. Comput.*, 2(4):399–408, 1986.

23. D. Kapur and P. Narendran. An equational approach to theorem proving in first-order predicate calculus. In *IJCAI*, pages 1146–1153. Morgan Kaufmann, 1985.
24. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.
25. M. Lauria and J. Nordström. Graph Colouring is Hard for Algorithms Based on Hilbert's Nullstellensatz and Gröbner Bases. In R. O'Donnell, editor, *CCC*, volume 79 of *LIPIcs*, pages 2:1–2:20. Schloss Dagstuhl, 2017.
26. J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.
27. M. Miksa and J. Nordström. A generalized method for proving polynomial calculus degree lower bounds. In *CCC*, volume 33 of *LIPIcs*. Schloss Dagstuhl, 2015.
28. A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *JSAT*, 9:53–58, 2014 (published 2015).
29. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
30. D. Ritirc, A. Biere, and M. Kauers. Column-wise verification of multipliers using computer algebra. In *FMCAD*, pages 23–30. IEEE, 2017.
31. D. Ritirc, A. Biere, and M. Kauers. Improving and extending the algebraic approach for verifying bit-level multipliers. In *DATE*, 2018, in press.
32. A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *DATE*, pages 1048–1053. IEEE, 2016.
33. A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using Gröbner bases. In *FMCAD*, pages 169–176. IEEE, 2016.
34. A. Tiwari. *An Algebraic Approach for the Unsatisfiability of Nonlinear Constraints*, pages 248–262. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
35. Wolfram Research, Inc. Mathematica, 2016. Version 10.4.
36. C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. Formal verification of arithmetic circuits by function extraction. *IEEE TCAD*, 35(12):2131–2142, 2016.
37. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, 2003.
38. E. Zulkoski, C. Bright, A. Heinle, I. S. Kotsireas, K. Czarnecki, and V. Ganesh. Combining SAT solvers with computer algebra systems to verify combinatorial conjectures. *J. Autom. Reasoning*, 58(3):313–339, 2017.